Microservicios con Java

@CertificaTIC

/Certificatic

www.certificatic.org

Contenido

Web Services - Arquitectura REST

Servicios RESTful

Los servicios web RESTful están diseñados para funcionar mejor en la web. Representational State Transfer (REST) es un estilo arquitectónico que especifica restricciones, como la interfaz uniforme, que si se aplica a un servicio web induce propiedades deseables, como rendimiento, escalabilidad y modificabilidad, que permiten que los servicios funcionen mejor en la web.

En el estilo arquitectónico REST, los datos y la funcionalidad se consideran recursos y se accede a ellos mediante **identificadores uniformes de recursos (URI).**, normalmente enlaces en la Web. Se actúa sobre los recursos mediante un conjunto de operaciones simples y bien definidas.

El estilo arquitectónico REST restringe una arquitectura a una arquitectura cliente / servidor y está diseñado para usar un protocolo de comunicación sin estado, típicamente HTTP. En el estilo de arquitectura REST, los clientes y los servidores intercambian representaciones de recursos utilizando una interfaz y un protocolo estandarizados.

Los siguientes principios fomentan que las aplicaciones RESTful sean simples, ligeras y rápidas:

- **Identificación de recursos a través de URI**: Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes. Los recursos se identifican mediante URI, que proporcionan un espacio de direccionamiento global para el descubrimiento de recursos y servicios.
- **Interfaz uniforme**: los recursos se manipulan mediante un conjunto fijo de cuatro operaciones de creación, lectura, actualización y eliminación: PUT , GET , POST y DELETE . PUT crea un nuevo recurso, que luego se puede eliminar usando DELETE . GET recupera el estado actual de un recurso en alguna representación. POST transfiere un nuevo estado a un recurso.
- **Mensajes** autodescriptivos: los recursos están desacoplados de su representación para que se pueda acceder a su contenido en una variedad de formatos, como HTML, XML, texto sin formato, PDF, JPEG,

JSON y otros. Los metadatos sobre el recurso están disponibles y se utilizan, por ejemplo, para controlar el almacenamiento en caché, detectar errores de transmisión, negociar el formato de representación adecuado y realizar la autenticación o el control de acceso.

- **Interacciones con estado a través de hipervínculos**: cada interacción con un recurso es sin estado; es decir, los mensajes de solicitud son independientes. Las interacciones con estado se basan en el concepto de transferencia de estado explícita. Existen varias técnicas para intercambiar estados, como reescritura de URI, cookies y campos de formulario ocultos. El estado se puede incrustar en los mensajes de respuesta para señalar estados futuros válidos de la interacción.

Puntos clave

A continuación, se enlistan los puntos más importantes para diseñar APIs Rest.

- Cliente – Servidor, Al separar las interfaces de usuario del manejo de los datos, se mejora la potabilidad de la interfaz atreves de múltiples plataformas, mejorando la escalabilidad de los componentes.

- Cache, Al agregar cache a ciertas operaciones mejorara el desempeño de ciertas consultas, sin tener necesidad de volver a realizar las consultas a la Base de Datos, es importante diseñar bien que operaciones tendrán cache u cuales no y cada cuando se refrescaran.

- Sistema en capas, Nos permite administrar mejor el comportamiento de cada uno de los componentes, de modo que cada componente no pueda ver más allá de la capa inmediata con la que interactúa.

- Versionado de APIs, esto nos ayuda a que si estamos haciendo un cambio en alguna de las APIS no dañe a los demás clientes que las están ocupando.

- Documentación, Cada uno de los cabios en la arquitectura del API deberá reflejarse en su respectiva documentación para saber qué cambios o fixes fueron los que se realizaron.

- Manejo de Errores, Incluir los mensajes de error los cuales proporcionaran el detalle respecto al error, es importante manejar tanto descripción de error como sus códigos.

Arquitectura con Api's.

Requisitos

A continuación, se enlistan los prerrequisitos a considerar para implementar una API Rest:

1. Agrupar las operaciones a realizar por modulo y con base a ese análisis realizar el llamado de las URIs raiz.

2. Definir los códigos de error y de éxito que manejara la API.

3. Definir lógica para versionado de las APIs.

4. Definir medios de autorización y autenticación de las operaciones que tendrá el API.

Documentación

Depende del lenguaje que se este utilizando siempre es importante mantener una documentación de todos los metodos que viven dentro de la aplicación, ya que la intención de un Web API es poder abrir nuestro sistema al consumo del mundo exterior cuidando siempre la seguridad e integridad de la información que expongamos.

El proceso de compartir nuestras APIs al mundo exterior es el objetivo principal de la creación de interfaces, el tener metodos para agilizar este proceso se vuelve indispensable, sin embargo, se genera un archivo de documentación aparte del código, se obliga a que en cada cambio integrado al sistema, se tenga que actualizar la documentación, lo que puede generar diferencias significativas entre lo que se tiene en código y lo que la documentación informa. En la actualidad existe un buen número de herramientas que nos ayudan a generar una buena documentación tomando en cuenta lo que vive en código de manera automática, lo que garantiza que no tendremos las diferencias antes mencionadas, una de las herramientas mas famosas se llama [Swagger](https://swagger.io/).

¿Pero que hace a Swagger tan especial? La respuesta es sencilla, swagger genera documentación que se expone dentro de la misma api y que únicamente nos va a reflejar lo que el código verdaderamente contenga, ademas de proporcionarnos un mecanísmo para poder ejecutar los metodos expuestos sin necesidas de alguna otra herramienta tipo Postman o SoapUI, por mencionar algunas

Diseño

El primer contacto con nuestras Web APIs es sin duda, la url base, se debe procurar mantener esta lo mas sencilla posible, se debe de considerar el siguiente patron:

Microservicios

- ApiGateway

  http[s]://ipaddress:port/api/{serviceName} –

- Services

  http[s]://ipaddress:port/{serviceName_service}/v1/{serviceName}

Monolítico

http[s]://ipaddress:port/api/v1/{serviceName}

Donde:

- ipaddress: dirección ip donde se encuentra desplegado el servicio
- port: puerto de escucha donde el servicio acepta peticiones
- api: nombre raíz de todo el servicio
- v1: version del servicio, se detallará mas adelante
- serviceName: nombre del servicio

Verbos Http (POST, GET, DELETE, PUT...)

Es mandatorio utilizar la nomenclatura correcta para definir el verbo http que utilizará cada método, existe una relación directa con los verbos y el acronimo CRUD (Create-Read-UpdateDelete) para entender rapidamente en que casos utilizar que verbo:

- POST [Create], se deberá utilizar para operaciones donde la información que enviemos de deba de persistir de alguna manera dentro de nuestro sistema ya sea porque la información se va a almacenar en alguna base de datos o será enviada incluso a algun servicio externo, , la información de envía dentro del Body de la petición.
- GET [Read], se deberá utilizar para todas las operaciones de consulta de información, es decir, su proposito es únicamente la obtención de información de parte de nuestro sistema y se pueden enviar los parametros de consulta en forma de QueryString, concepto que se definirá mas adelante.
- DELETE [Delete], se deberá utilizar para enviar registros que van a ser borrados de manera lógica o física de manera indistinta y las condiciones de los registros a eliminar se enviarán mediante QueryString.
- PUT [Update], se deberá utilizar para hacer la actualización de datos que ya estan almacenados dentro del sistema y que únicamente requieren un cambio de contenido, la información de envía dentro del Body de la petición.

Nombrado de métodos

El nombrado de los métodos es un proceso de suma importancia puesto que en unos pocos caracteres debemos de representar el objetivo de nuestro método sin la necesidad de que el consumidor tenga que entrar a revisar la lógica implementada o incluso la documentación a detalle. No debemos olvidar que el nombrado va de la mano con los verbos http, por ejemplo, si queremos exponer un metodo que obtenga los usuarios de nuestro sistema, se tendrían las siguientes opciones:

Incorrecto:

- GET /getUsers

- GET /getAllUsers

- GET /ObtenerUsuarios

Correcto:

- GET /users

En el ejemplo el uso de *GET /getUsers*, es repetitivo colocar un get cuando el método ya es un get, en el caso de *GET /getAllUsers*, si un método no tiene condiciones de búsqueda se debe considerar como un retorno de toda la información disponible y para el último caso *GET /ObtenerUsuarios*, se tiene que evitar en nombrado en español, siempre debe ser en ingles.

Para el caso de *GET /users* sin entrar en mas detalles se entiende que el método nos regresará todos los usuarios, si utilizamos el mismo nombre para las operaciones CRUD tendríamos lo siguiente:

Correcto:

- GET /users

- POST /users

- PUT /users

- DELETE /users

Incorrecto:

- GET /getUsers

- POST /saveUsers

- PUT /updateUsers

- DELETE /deleteUsers

Manejo de errores

Los errores de sistema deben ser alineados con los estatus de error de HTTP, para que cualquier consumidor de nuestra API pueda generar validaciones limpias dentro de los sistemas que invocarán nuestra API, teniendo que cualquier respuesta diferente a un código HTTP 200 es sin duda un error.

El manejo de errores hace mas facíl la ubicación de problemas a nivel de capaz, ya que se cuenta con estas 3 grupos de códigos:

- Todo funcionó de manera correcta [Success] (Http status 200+)
- La aplicación que invoca ejecuto algo de manera incorrecta [Client error] (Http status 400+)
- El servidor no pudo procesar la petición de manera correcta [Server error] (Http status 500+)

Versionamiento

El versionamiento es una de las partes mas importantes al momento de diseñar una WebAPI, debido a que en la actualidad multiples sistemas pueden consumir nuestras interfaces, y no necesariamente todos estos sistemas van a poder adaptarse a los cambios que hagamos en nuestro servicio, un ejemplo claro es cuando desarrollamos APIs para dispositivos móviles, no necesariamente todos los usuarios van a poder actualizar para apuntar a nuevas versiones de los servicios expuestos, es por eso que por un tiempo debemos dejar convivir a más de una versión de nuestros métodos.

Desarrollo de microservicios (spring boot).

Puntos clave

A continuación se enlistan los puntos mas importantes para implementar microservicios.

- Durante la fase de diseño de los microservicios, se debe establecer “Boundarys” de negocio(dominio), esto permite clarificar las responsabilidades, limites de cada uno de los dominios y da una pauta para segregar los componentes.

- Utilice el Principio de responsabilidad única (SRP): tener un alcance comercial limitado y enfocado para un microservicio nos ayuda a cumplir con la agilidad en el desarrollo y la entrega de servicios.

- Asegúrar que el diseño de microservicios garantice el desarrollo ágil / independiente y la implementación del servicio.

- Enfocarse en el alcance del microservicio (Responsabilidad), pero comprender que no se trata de hacer los servicios más pequeños. El tamaño correcto del este será el tamaño requerido para facilitar una capacidad de negocio requerida.

- A diferencia en SOA, un microservicio dado debe tener muy pocas operaciones / funcionalidades y un formato de mensaje simple.

- A menudo es una buena práctica comenzar con componentes satelites relativamente amplios para refactorizar a un tamaño reducido (según los requerimientos de negocio).

- Definir la modularidad con base al dominio de negocio, esto permitirá empaquetar productos, ser flexibles cuandos se requiera escalar o desacoplar.

- Definir equipos pequeños de trabajo, entré mas persónas se requieran en un equipo existirá el overhead de gestionarlas, equipos pequeños son agiles y escalables para los productos.

- Adopción de nuevas “Best practices”, el software su naturaleza es el “cambio”, adoptar que todo evoluciona nos permitre prepararnos para estos cambios de una forma más agil para responder las necesidades por el mercado.

- Contemplar la flexibilidad de soportar diferentes lenguajes de programación, ya que nos permiten solucionar los problemas con los artefactos necesarios.

- Capacidad para deplegar/desinstalar independientemente de otros microservicios, así mismo debe poder escalar en cada nivel de los microservicios. Esto nos ayuda a que si un servicio falla no afecte ninguno de los otros servicios.

- Realizar un análisis y diferenciar claramente entre las funciones de su negocio y sus servicios. Ya que esto resultaría en una fragmentación excesiva de su arquitectura.

- Para obtener un valor óptimo de estos microservicios, debe automatizar la gestión de compilación y despliegue, por lo tanto, necesitar un buen conjunto de herramientas de DevOps.

Lenguajes, API’s y/o Frameworks

1) Java 11+

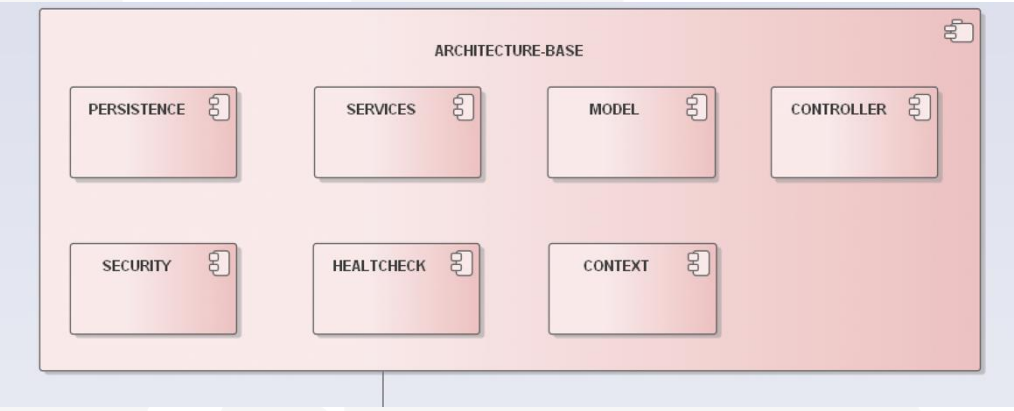
a) Es un lenguaje de programación orientado a objetos, creado por Sin Microsystems, usado generalmente para el desarrollo de servicios backend, web, etc.
b) Java es considerado como una plataforma la cual puede ejecutar cualquier lenguaje que cumpla con las especificaciones del bytecode.
c) Las nuevas versiones de Java, estan diseñadas modularmente, lo que permiten una gestión de memoria mas efectiva y minimizadas para ambientes cloud
d) El tiempo inicializaciòn esta mejorado para este tipode aplicaciones que mejoran el tiempo de disponibilidad de una aplicativo

2) Spring Framwork

a) Spring es un ecosistema que permite desarrollar y desplegar servicios java, contiene una gran variedad de componentes en los aspectos transversales, (Seguridad, Batch, Integration, WebFlow, Rest) que agilizan el desarrollo y la puesta en producción de los productos.

b) Par entornos cloud, utilizaremos Spring boot, el cual proporciona un contenedor con las minimos requerimientos.

Arquitectura Base



A continuación se describen los componentes recomendados que debe integrar una arquitectura base de microservicios.

- PERSISTENCE

a) Capa de persistencia que contendrá toda la responsabilidad de acceso a los datos considerando base de datos relacionales, documentales, cloud storage, se deben definir cada una de las funcionalidades de negocio que puedan ser comunes o abstractas y sea en un acelerador para los productos.

- SERVICES

a) Capa de servicios que contendrá aquellas funcionalidades “base” y ofrecer requerimientos trasnversales como , bitacoreo, logging, seguridad, o algún aspecto técnico.

- MODEL

a) Capa de entidades que define el esquema de datos que se manipulará durante la ejecución de flujos, agregando aquellos datos base o de control para el negocio, como fecha de actualización, usuario que afecto, borrado lógico.

- CONTROLLER

a) Capa de exposición que establece las acciones base para interactuar con otros puntos de conectividad ya sea REST, SOA.

- SECURITY

a) Capa de Seguridad que permite obtener aquellos datos de una sesión, de un usuario, de una transacción que se esta operando en el flujo de negocio, para

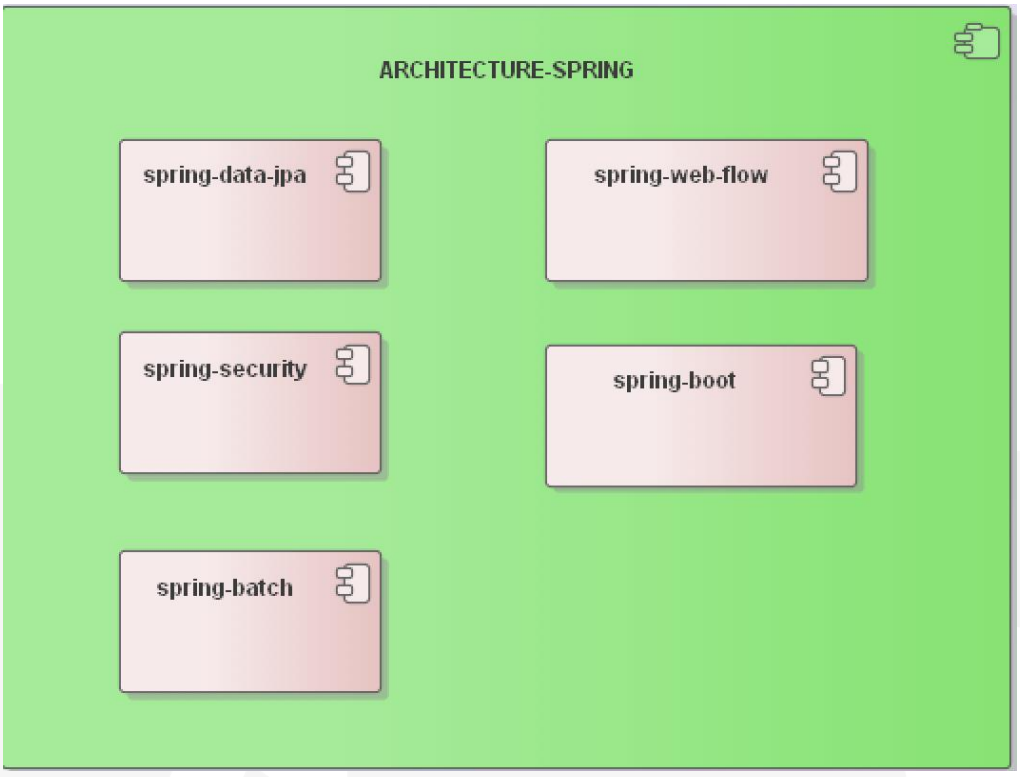
poder restringirla, autorizarla en niveles de aplicativo, no sustituye a herramientas que existan en el negocio, solo permitira integrarse.

- HEALTCHECK

a) Capa para los microservicios que permite definir un healtcheck de negocio, que indicará que el servicio esta listo para operar recibir el trafico o indicará que aun esta operando, se definirán elementos base para solo implmentar

Arquitectura Spring

A continuación se especifican algunos de los componentes que se pueden implementar con Spring framework en su ecosistema para los microservicios.



- Spring Data (JPA)

a) *"La misión de Spring Data es proporcionar un modelo de programación familiar y consistente basado en Spring Framework para el acceso a los datos, al tiempo que conserva los rasgos especiales del almacén de datos subyacente. Facilita el uso de tecnologías de acceso a datos, bases de datos relacionales y no relacionales, marcos de reducción de mapas y servicios de datos basados en la nube. Este es un proyecto general que contiene muchos subproyectos que son específicos de una base de datos determinada. Los proyectos se desarrollan trabajando en conjunto con muchas de las compañías y desarrolladores que están detrás de estas emocionantes tecnologías"*.
https://spring.io/projects/spring-data

- Spring Security

a) "Spring Security es un marco de autenticación y control de acceso potente y altamente personalizable. Es el estándar para garantizar aplicaciones basadas en Spring.

Spring Security es un marco que se centra en proporcionar autenticación y autorización a las aplicaciones Java. Al igual que todos los proyectos de Spring, el verdadero poder de Spring Security se encuentra en la facilidad con que se puede extender para cumplir con los requisitos personalizados"

https://spring.io/projects/spring-security

- Spring Batch

a) "Un marco de trabajo por lotes, ligero e integral diseñado para permitir el desarrollo de aplicaciones robustas vitales para las operaciones diarias de los sistemas empresariales.

Spring Batch proporciona funciones reutilizables que son esenciales en el procesamiento de grandes volúmenes de registros, incluidos el registro / rastreo, la gestión de transacciones, las estadísticas de procesamiento de trabajos, el reinicio de trabajos, la omisión y la gestión de recursos". https://spring.io/projects/spring-batch

- Spring Boot

a) *"Spring Boot facilita la creación de aplicaciones independientes basadas en Spring de grado de producción que puede 'simplemente ejecutar'. Tomamos una visión obstinada de la plataforma Spring y las bibliotecas de terceros para que pueda comenzar con un mínimo de alboroto. La mayoría de las aplicaciones Spring Boot necesitan una configuración mínima de Spring".* https://spring.io/projects/spring-boot

Desarrollo de microservicios (Api manager).

Api Manager

Los microservicios son API detalladas, lo que significa que las aplicaciones cliente deben interactuar con múltiples servicios si los clientes se comunican con los microservicios directamente a través de mecanismos como REST basado en HTTP, Thrift o gRPC (para comunicación síncrona basada en solicitudes / respuestas). Este enfoque, conocido como comunicación directa de cliente a microservicio, producirá problemas considerables como los siguientes:

Introduzca el espagueti de punto a punto que SOA intentó eliminar con el uso de un bus de servicio empresarial.

Si hay discrepancias entre las necesidades del cliente y las API detalladas expuestas por cada uno de los microservicios, el cliente tendrá que realizar varias llamadas para obtener lo que necesita. En una aplicación compleja, esto podría significar cientos de llamadas de servicio.

Consumir servicios a través de interfaces y protocolos no uniformes.

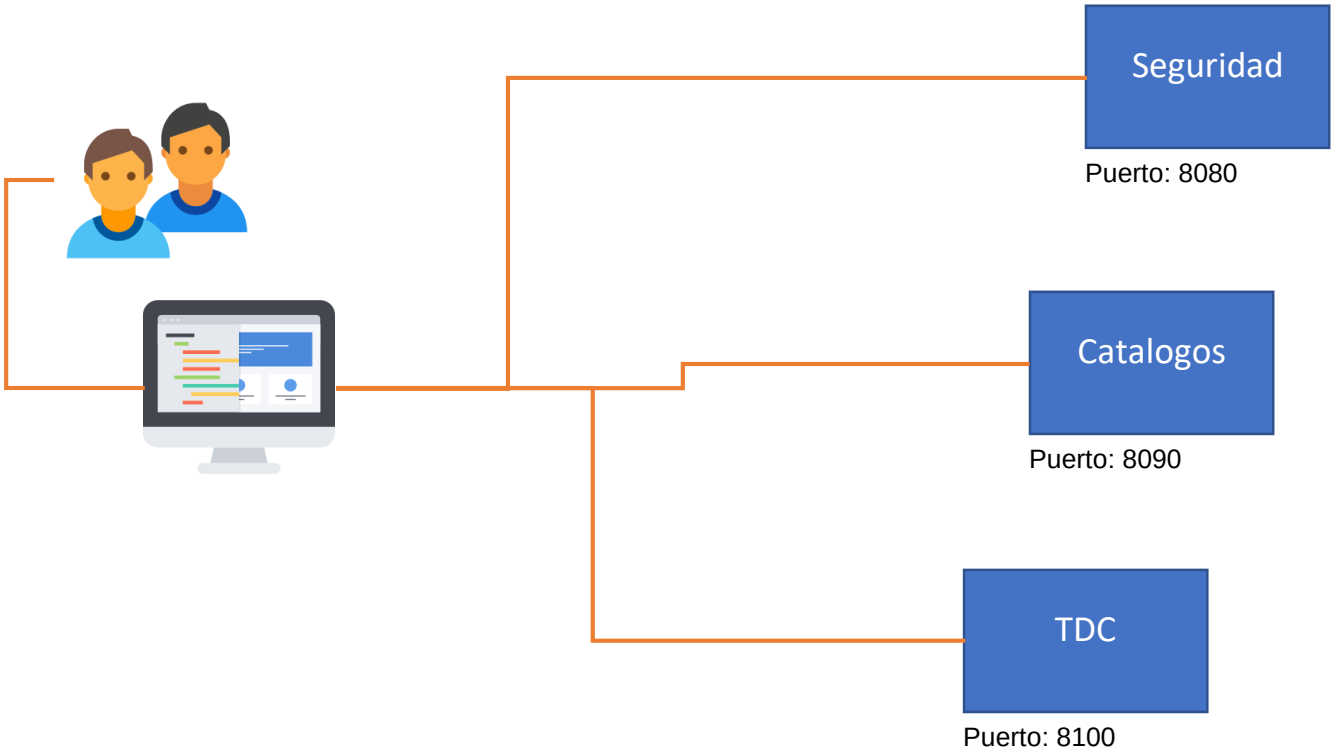
Refactorizar los microservicios será un desafío si los clientes se comunican con los servicios directamente.

Debido a este tipo de problemas, rara vez tiene sentido que los clientes hablen directamente con los microservicios. Al diseñar y construir aplicaciones basadas en microservicios grandes o complejas, adoptar el patrón API Gateway es un buen enfoque. La implementación de una puerta de enlace API como punto de entrada único para todos los clientes abordará los problemas que se enfrentan en la comunicación directa entre el cliente y el microservicio. Todas las solicitudes de los clientes pasan primero por API bien definidas en una puerta de enlace de mensajes liviana (también conocida como API Gateway). La puerta de enlace es principalmente responsable del enrutamiento de solicitudes, la composición y la traducción del protocolo. Las API de API Gateway a menudo manejarán una solicitud invocando varios microservicios y agregando los resultados.

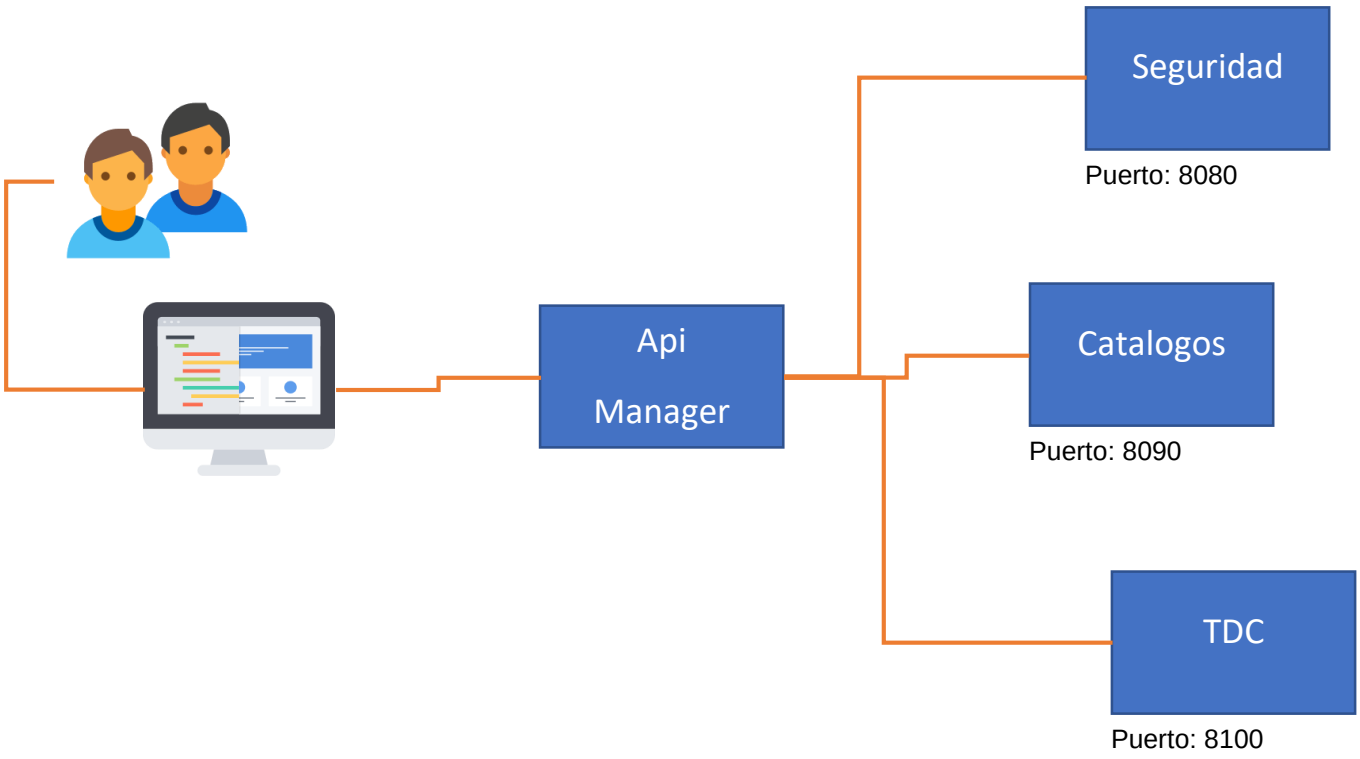
Una plataforma de administración de API debe proporcionar formas de incorporar y administrar a los desarrolladores, crear un catálogo y documentación de API, generar informes de uso de API, producir o monetizar las API y hacer cumplir la limitación, el almacenamiento en caché y otras precauciones de seguridad y

confiabilidad. API Manager proporciona las herramientas, el control y la visibilidad para escalar microservicios a través de API a nuevos desarrolladores y conectarlos a nuevos sistemas.

Sin api Manager



Con api Manager



Puntos clave

Simplificación del código de cliente

Control de acceso y seguridad

Rate Limiting

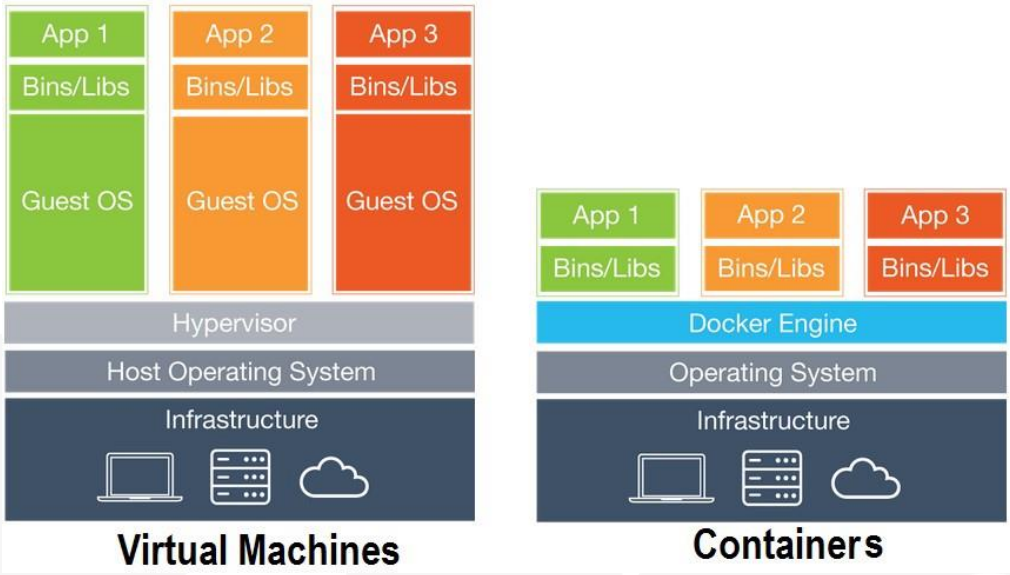
Administración de microservicios como APIs

Flexibilidad de despliegues

Desarrollo de microservicos (docker kubernates).

Contenedores

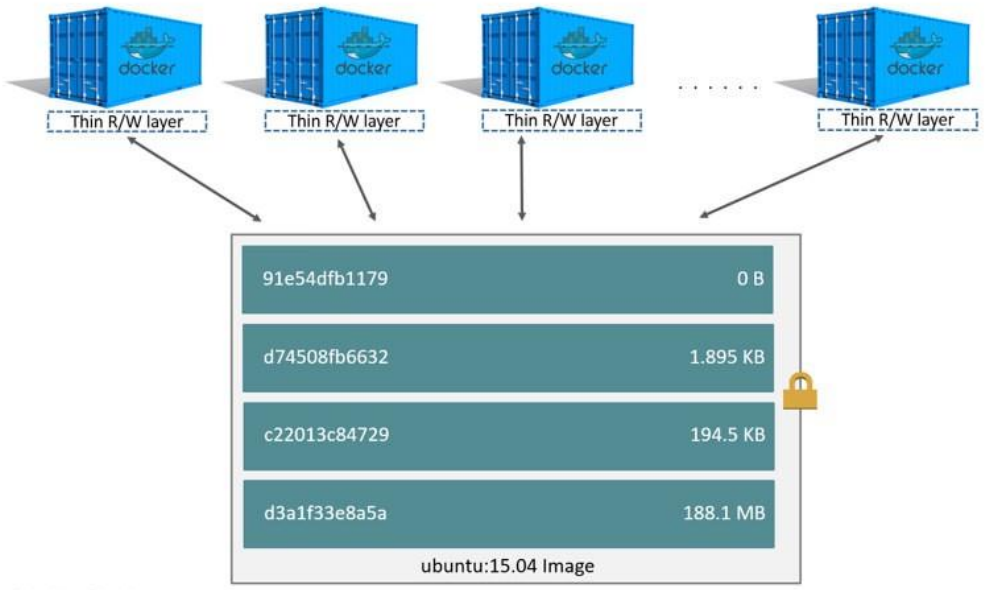
Es una pieza de software liviana, independiente, empaquetable y ejecutable que incluye todo lo que necesita para correr: código, runtime, herramientas de sistema, librerías y configuraciones' traducido de la página oficial de docker, eso traducido a lenguaje práctico significa que es un pedacito de software que puede ejecutarse por si mismo sin necesidad de nada más. Un proceso que se aloja en la memoria de un anfitrión y que idealmente copia sus características y las extiende para obtener la funcionalidad necesaria. Muy distinto a las máquinas virtuales, pues ellas tienen una copia completa del sistema operativo, mientras que los contenedores solo incluyen las diferencias con respecto al anfitrión



Los beneficios son que podemos explotar de mejor manera los recursos de computo, ya que no se necesitan máquinas virtuales completas para gestionar aplicaciones.

Kubernetes

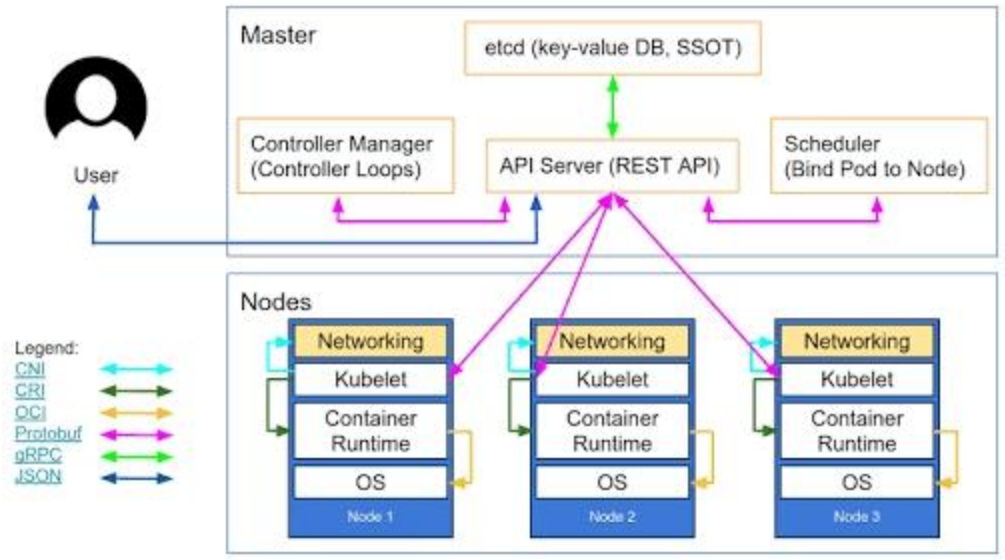
De primera instancia, los Microservicios se expresan en forma de Dockers para poder simplificar su despliegue y atomicidad. Pero llevando este tipo de soluciones a producción nos daremos cuenta que se complica el despliegue.
En el siguiente diagrama mustra una palicación real, en la que se tienen multiples microservicios, pero gestionados de forma aislada:



A pesar de que parece sencillo, nos podemos dar cuenta que este tipo de arquitecturas es complicado gestionarlas, para eso nacen productos orquestadores como Kubernates.

Entre las labores de un orquestador está la labor de levantar y aprovisionar servidores, monitorear y configurar la red, escalar servicios, volúmenes y todas las demás configuraciones necesarias para que las aplicaciones estén siempre disponibles. La idea detrás de todo esto es automatizar al máximo las operaciones, para minimizar el factor humano y tener un control mas holístico de toda la infraestructura de red.

El siguiente diagrama muestra un despliegue de Kubernates:



En esta arquitectura podemos notar que la distribución de los nodos de Kubernates y la gestión de los contenedores hacen que haya alta disponibilidad y seguridad en los recursos, a la vez, que se puede realizar un auto escalamiento en caso de existir una alta transaccionalidad.

Administración de contenedores.

Los contenedores son herramientas que simulan un entorno de Sistema Operativo y nos permiten desplegar nuestras aplicaciones sin necesariamente preocuparnos (demasiado) por los diferentes sistemas de configuración.

Al exponerse a cualquier tecnología de contenedores, lo primero con lo que interactuar es probablemente una imagen de contenedor. Este es un paquete binario que contiene todos los archivos necesarios para ejecutar una aplicación dentro de un contenedor.

Podemos decidir crear una imagen desde cero desde nuestras máquinas locales o decidir extraer una de un registro de imágenes. De cualquier manera, puede ejecutar la imagen para producir una aplicación en ejecución dentro de un contenedor.

Construcción

A Dockerfilees un archivo de texto que contiene comandos especificados por un usuario (un desarrollador) al crear una imagen.

Veamos el siguiente ejemplo al crear una imagen ligera:

```
FROM alpine
MAINTAINER <your-name> <your-email>
COPY <application-directory> <destination>
ENTRYPOINT ["<entrypoint>"]
```

La primera línea indica que estamos creando el contenedor a partir de la imagen más delgada disponible.

La segunda línea solo muestra los detalles del autor: nombre y correo electrónico.

La tercera línea copia los archivos del directorio de la aplicación en su sistema al directorio de la aplicación en el contenedor de Docker.

La última declaración contiene el comando (ENTRYPOINT) que se utiliza para iniciar la ejecución de la aplicación en el contenedor desde donde ha señalado la aplicación.

Luego podemos crear la imagen usando el formato de comando a continuación:

```
docker build -t <image-name>:<image-version> .
```

```
docker build -t sectionio-image:2.0 .
```

Ejecución

Docker tiene una herramienta CLI para implementar sus contenedores.

Aquí hay una sintaxis de comando de ejemplo para ejecutar una imagen:

```
docker container run --publish 8080:80 <image-name>
```

Después de iniciar la imagen, asigna el puerto 8080 en nuestro localhost al puerto 80 en nuestro contenedor.

Limpieza

```
$ docker rmi <tag-name>
```

O

```
$ docker rmi <image-id>
```

Mejores practicas

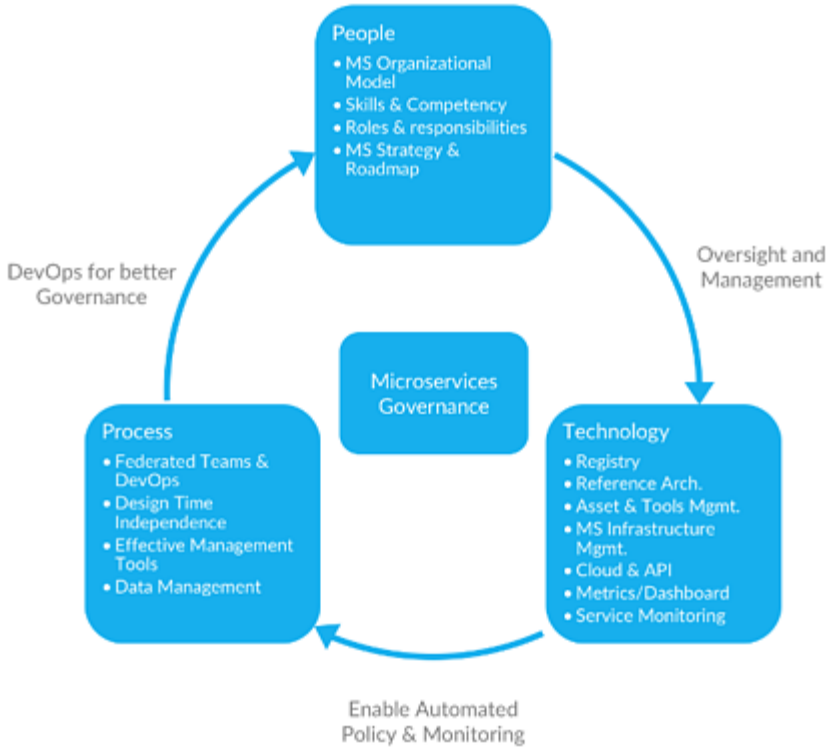
- No ejecute contenedores Docker con acceso de nivel raíz. Utilice la etiqueta -u al principio de cada contenedor, que dará acceso predeterminado al usuario, en lugar del administrador.
- No almacene las credenciales de Docker dentro del contenedor. En su lugar, use las variables ambientales de Docker para administrar las credenciales, de modo que una brecha en un contenedor no sea una bola de nieve.
- Verifique y administre los privilegios de tiempo de ejecución y las capacidades de Linux. La configuración predeterminada para los contenedores nuevos es sin privilegios, lo que significa que no podrán acceder a ningún otro dispositivo. Para los contenedores que requieren colaboración, aplique la etiqueta -privileged, aunque esto permitirá el acceso a todos los dispositivos.
- Utilice herramientas de seguridad. Las herramientas de Docker que escanean imágenes en busca de vulnerabilidades de seguridad conocidas pueden aumentar considerablemente la seguridad.
- Considere los registros privados. Docker Hub pone a disposición una amplia gama de opciones de registro gratuitas y compartidas. Pero muchas empresas no se sienten cómodas con el acuerdo de seguridad que esto representa y eligen alojar sus propios registros o recurrir a servicios de repositorio de artefactos en las instalaciones como JFrog Artifactory. Evalúe sus necesidades de seguridad con su equipo y decida si los registros públicos funcionarán para usted.

Gobierno de servicios y microservicios.

El gobierno de microservicios es una metodología o enfoque que establece políticas, estándares y mejores prácticas para la adopción de microservicios a fin de habilitar un entorno de TI empresarial ágil.

La gobernanza con monolitos está centralizada. Las decisiones se toman de arriba hacia abajo, se mantiene un control rígido para garantizar que los estándares estén en su lugar en toda la organización y dentro de la pila de aplicaciones. Con el tiempo, este modelo degenera, creando un sistema arquitectónico y tecnológico estancado, minimizando así la innovación.

Los microservicios promueven una modelo políglota con respecto al stack de tecnología para lenguajes, herramientas y almacenes de datos compatibles. El concepto principal de estos microservicios es la reutilización de activos y herramientas que pueden descentralizarse. El tema central de la gobernanza de la descentralización es el concepto de construirlo y ejecutarlo. Este modelo descentralizado es el más adecuado para las gobernaciones de microservicios. Los beneficios de la gobernanza descentralizada brindan a los equipos de microservicios la libertad de desarrollar componentes de software utilizando diferentes pilas.



Documentación de servicios

A continuación se enlistan los prerequisitos a considerar para implementar un microservicio:

- Diseñe una parte del sistema de software para reflejar el modelo de dominio de una manera simple, esto permitirá definir un solo lenguaje para todo el equipo de trabajo.
    1. Se considera utlizar diagramas UML (Actividades, Secuencia, Clases, Datos)

- Se recomiendo realizar un diseño de isolación de negocio vs el dominio de datos, con diagramas de flujo y modelo de dominio de datos, esto permitirá tener la claridad sobre

- Diseñar los componentes con la aplicación del patron de diseño “Single Responsability”, Alta cohesión y Bajo Acomplamiento, ya que esto permitirá definir Clases, Interfaces, Metodos, Schemas.

- Definir los eschemas de datos que cada una de las funciones o requerimientos, con la notación de Entidad Relaciona y no de SQL o NOSQL, ya que la definición de una tecnología aun depende del entendimiento de problema.

- Diseñar siempre con Interfaces y no con implementaciones, las interfaces definen los contratos para los instancias, lo cual permitirá cambiar la implementación sobre un contrato ya definido

- Definir métricas sobre la concurrencia de usuarios, volumetría de datos, esto permitirá ayudar a dividir o agregar mas componentes que permitán cubrir los aspectos no funcionales.

- Sujetarse a estandares de lenguaje de programación de la industría, cada versión agregar nuevas caracteristicas que facilitan o deprecan funcionalidades.

- Los microservicios se pueden configurar estilo punto a punto, toda la lógica del envío de mensajes reside en cada punto final y los servicios pueden comunicarse directamente. Cada servicio expone un API REST y un servicio invoca a otro através de su API REST.

- Los microservicios se pueden configurar estilo API Gateway, el cual se basa en usar una Gateway liviana de mensajes como el punto de entrada principal para todos los clientes.

- Los microservicios se pueden configurar estilo Message Broker, el cual se basa en integrar una mensajería asíncrona, como solicitudes unidireccionales y mensajes de publicación-suscripción mediante queues o topics.

- Dado que los servicios pueden fallar en cualquier momento es importante detectar y/o monitorear las fallas rápidamente y si es posible restaurar los servicios automáticamente, para esto existen algunos patrones de diseños como:

    a. Circuit Breaker

b. Bulkhead
c. Timeout

A continuación se enlistan los documentos a considerar para implementar un microservicio:

- Diagrama de Dominio Total
- Diagrama de Actividades
- Diagrama de Dominio de Datos
- Diagrama de Clases
- Diagramas de secuencia.
- Metricas de Concurrencia, Volumetría

Portafolio de servicios

Las organizaciones están adoptando cada vez más una mentalidad centrada en los resultados para atender a los clientes internos y externos. Esta mentalidad se manifiesta en forma de productos y servicios físicos o digitales. Permite un mayor enfoque en la creación de productos mínimos viables (MVP) que logren los objetivos comerciales y mejoren la satisfacción general. Esto es especialmente cierto para las organizaciones de tecnología de la información (TI) que ven a los proveedores externos convertirse en los asesores preferidos de los usuarios internos.
Para lograr esta mentalidad centrada en los resultados, la gestión de la cartera de servicios está surgiendo como un enfoque integrado para fusionar los equipos, los objetivos comerciales y de TI, y los canales relacionados que respaldan una capacidad discreta.
Mediante el uso de roles como los gerentes de productos o de ofertas, y herramientas y técnicas como el pensamiento de diseño empresarial, las construcciones de equipos ágiles y los mercados con catálogos, su organización puede crear una diferenciación competitiva.

El cambio a la nube introduce nuevas características, como la facturación basada en el consumo y el autoservicio. Estas características están cambiando la forma en que los usuarios solicitan, usan y mantienen las capacidades. Los usuarios solicitan experiencias similares a las de las compras de los consumidores en las que pueden usar catálogos y carritos de compras para solicitar nuevos productos y servicios.
Este cambio brinda una oportunidad para que los equipos de TI empresariales sigan siendo relevantes para el negocio y demuestren valor a los usuarios mediante la implementación de escaparates y catálogos digitales. A través de esos escaparates y catálogos, ofrecen capacidades que los usuarios pueden solicitar y aprovisionar según sea necesario. Las capacidades pueden proporcionarse directamente a través de un modelo de servicio en la nube oa través de terceros con capacidades de valor agregado, como el monitoreo o el cumplimiento de los estándares organizacionales a través de un modelo de intermediario en la nube. En última instancia, las capacidades se basan en la cartera de servicios.

Un paso clave en la definición de una cartera de servicios es comprender qué hace bien su organización. Para comprender qué hace bien su organización, haga estas preguntas:

- ¿Cuáles son nuestros principales servicios?
- ¿En qué áreas nos enfocaremos o nos especializaremos?
- ¿Cómo se correlacionarán nuestros servicios principales con nuestros objetivos estratégicos?
- ¿Qué capacidades son las más populares y cómo se utilizan?
- ¿Cuáles son los deseos y necesidades del cliente?

Después de responder a esas preguntas, puede comenzar a identificar las áreas de enfoque que puede asignar a las capacidades que ofrece actualmente y las futuras. Para identificar más áreas de enfoque, utilice modelos de madurez y aportes del usuario a través de Enterprise Design Thinking. Después de identificar todas las capacidades posibles, priorícelas por valor y demanda para desarrollar una cartera holística.
Para las organizaciones que llevan a cabo la planificación de la arquitectura empresarial, estas nuevas actividades pueden parecer similares, pero una lección clave es que es posible que sus capacidades deban aumentarse con una vista de la

perspectiva holística del usuario y del negocio. Esta vista adicional lo ayuda a evitar el desafío común en el que una organización lanza un servicio solo para ver un uso limitado por parte de la audiencia objetivo.
Al introducir roles multidisciplinarios, es más probable que las carteras de servicios cumplan con los objetivos comerciales y de TI. Dichos roles incluyen gerentes de cartera que son responsables de la estrategia comercial y técnica general de todos los productos y servicios físicos o digitales y gerentes de oferta o productos que están a cargo de capacidades individuales específicas.

Lineamientos de diseño

A continuación, se enlistan los prerrequisitos a considerar para implementar un API Rest:

- Se tiene que tomar en cuenta los siguientes arquetipos:
- DocRoot
  Es el recurso raíz de un API REST y generalmente es el punto de entrada a una API.
  Por ejemplo:
  https://organizacion.mx

- Resource
  Es un concepto que asimila a una instancia de objeto o registro de base de datos.
  Por ejemplo:
  https://organizacion.mx/user/alert/2345
  https://organizacion.mx/user/76

- Collection
  Es un directorio de recursos administrado por el servidor, los clientes pueden proponer nuevos recursos para agregar a una colección.
  Por ejemplo:
  https://organizacion.mx/saldo

https://organizacion.mx/saldo/mexico/11

- Controller
  Modela un concepto de procedimiento, son como funciones ejecutables, con parámetros y valores de retorno.
  Por ejemplo:
  POST /saldo/mexico/enviar

- La estructura del nombrado de las URL se recomienda lo siguiente:
  a. Crear URLs las cuales sean fácil de leer para consumo humano y para la construcción de las aplicaciones, utilizando sustantivos que hagan referencia a la operación que se está diseñando.

  b. Evite anidar relaciones de recursos secundarios más de una vez, en su lugar seleccione una ruta de recursos raíz.

  c. El uso de URL como parámetros es un patrón de diseño común, por ejemplo:
     i. https://organizacion.mx/images?valor=347

  d. Evite crear relaciones de recursos primarios / secundarios anidados, las rutas podrían quedar profundamente anidadas y esto llevaría a una difícil comprensión de la operación diseñada, por ejemplo:
     i. /cliente/{client_id}/office/{office_id}/deparment/{deparment_id}

  e. Considere exponer una URL que contenga un identificador único y estable para todos los recursos y así tener agrupadas las operaciones por módulos o funcionalidades.

  f. Evite utilizar métodos CRUD HTTP o verbos en las rutas URL, por ejemplo: GET, DELETE, etc.

- Los siguientes son algunos de los metods RESTFUL soportados:
  a. GET

b. PUT
c. DELETE
d. POST
e. PATCH

- Tanto en los Request como los Reponse se pueden agregar Headers algunos de ellos son los siguientes:
  a. Request
     i. Authorization
     ii. Date
     iii. Accept
     iv. Content-Type
     v. Accept-Encoding
     vi. Accept-Charset

  b. Response
     i. Date
     ii. Content-Type
     iii. Content-Encoding
     iv. Preference-Applied
     v. X-API-Version
     vi. X-API-Warn

- Se puede agregar y/o configurar el almacenamiento en cache para ciertas operaciones donde los datos no cambian continuamente, esto nos puede ayudar a obtener más rápido los datos.

- El uso de Cache nos puede traer errores ya que nos puede traer datos desactualizados, por eso es importante definir a que operación agregar cache y cada cuando se actualizarían los datos.

- En cada una de las peticiones especificar la versión como parte del tipo de medio en el encabezado de la petición ya que con esto se tiene un mejor control de la funcionalidad a invocar.

- Evite representar campos de cadena sin valor como “”. Devuelva un valor NULL en su lugar, ya que representa diferentes cosas.

- Evite regresar NULL para arreglos vacíos. En su lugar devuelva un arreglo vacío [] ó {}.

- Formatee booleanos como booleanos (sin cadenas), y nulos como nulos (sin cadenas).

- Considere limitar el número de recursos devueltos en una respuesta a un subconjunto (paginado) de todo el conjunto disponible, dado que al realizar un paginado, el desempeño del servicio no se ve afectado en peticiones de grandes volúmenes.

- Considere proporcionar enlaces para enviar un conjunto de datos paginados, por ejemplo:
    a. GET /artículos?pagina[numero]&pagina[tamanio]

- Ordene los valores NULL como menos importantes, para los servicios que admiten ordenación.

- Incluya un encabezado de respuesta X-API-Warm cuando se solicita una versión de API obsoleta en el encabezado.

- Indique en la documentación de la API cuando las versiones obsoletas de la API llegan a su final de vida útil.

- Aplique HTTPS en todos los endpoints, recursos y servicios.

- No expongas más de lo que se debe, si cierta propiedad de un objeto es interna de su negocio y no es útil para un consumidor no la devuelva.

Mejores practicas

- Práctica n. ° 1: configure el contexto de seguridad para sus pods y contenedores
  Al diseñar pods y contenedores, una cosa que debemos tener en cuenta es aplicar el contexto de seguridad a sus pods y contenedores. Es una

propiedad definida en la implementación yaml. Los parámetros de seguridad para el pod o contenedor están controlados por el contexto de seguridad. Por ejemplo, podemos contexto de seguridad como: SecurityContext-> runAsNonRoot.
Además, el control de admisión "DenyEscalatingExec" se puede usar para denegar la ejecución y adjuntar comandos a los pods si está ejecutando pods con privilegios escalados.

- Práctica n. ° 2: asegúrese de que solo se usen imágenes autorizadas en el entorno
  Un proceso para verificar y garantizar que solo las imágenes que se adhieren a la política del área de organización se usen y se les permita ejecutar en su entorno. Sin este proceso, existe un riesgo significativo de ejecutar contenedores maliciosos o vulnerables. No ejecute imágenes que se descargan de fuentes desconocidas o no confiables. Puede ser peligroso para la organización.
  Las imágenes aprobadas deben almacenarse en registros privados. Asegúrese de que estos registros privados solo almacenen modelos aprobados y no imágenes no confiables. Antes de enviar la imagen al registro individual, se debe escanear una imagen construida para detectar vulnerabilidades. Solo las imágenes, que no tienen problemas ni debilidades, deben venderse al registro. Se debe realizar una canalización de CI para integrar la evaluación de seguridad. Con esta canalización de CI como parte del proceso de construcción, podemos asegurarnos de que solo el código aprobado para producción se use para la operación de imágenes de construcción.
  Si se produce un error en la evaluación de seguridad, debe fallar el proceso de la canalización, de modo que las imágenes que son vulnerables y no aprobadas después de la evaluación no se envíen al registro de imágenes.
- Práctica # 3 - Traslado de implementaciones a producción
  Las organizaciones deben verificar las vulnerabilidades antes de trasladar las implementaciones a producción, de lo contrario, con la migración de implementaciones a producción aumentan las cargas de trabajo vulnerables. Además, se debe mantener una cultura DevOps / DevSecOps saludable en la organización para evitar estos problemas.
- Práctica n.º 4: asegúrese de asegurar las canalizaciones de CI / CD en Kubernetes

Asegúrese de tener una canalización de CI / CD ejecutándose en el entorno construido, que desarrolla, prueba e implementa las cargas de trabajo antes de usarlas en el clúster de kubernetes. Además, CI / CD debe configurarse para informar rápidamente de posibles vulnerabilidades a los desarrolladores. De lo contrario, si estas imágenes potencialmente vulnerables se envían al clúster de producción de kubernetes, los atacantes pueden explotar estas vulnerabilidades y obtener acceso a los grupos. Para evitar este problema, el código para las configuraciones de implementación debe verificarse a intervalos regulares.

- Práctica # 5 - Uso de políticas de autorización
  La complejidad surge cuando hay múltiples dispositivos y microservicios interconectados con muchos usuarios diferentes; esto es una pesadilla. Kubernetes ofrece varios métodos de autorización. Deberíamos usar el Control de acceso basado en roles (comúnmente conocido como RBAC) o el Control de acceso basado en atributos (ABAC) que garantizará que ningún usuario tenga más permiso del que necesita para realizar su tarea.
- Práctica # 6 - Crear segmentación de red entre contenedores. Si ejecutamos diferentes aplicaciones en un clúster de kubernetes, existe un riesgo considerable de que si uno de los formularios se ve comprometido, el otro también se convierta en un objetivo fácil. La segmentación de red puede ser la solución a este problema, ya que garantiza una comunicación segura entre contenedores. Los contenedores solo pueden comunicarse con los que deben. Pero la tarea de crear segmentación de red entre contenedores no es una tarea fácil, debido a la naturaleza "dinámica" de las IP de los contenedores. Los firewalls de red se pueden usar para evitar la comunicación entre clústeres si es necesario.

Certificatic

Beyond Certification

www.certificatic.org